Game Portability Using a Service-Oriented Approach

Ahmed BinSubaih & Steve Maddock
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield, U.K. +44(0) 114 2221800
{a.binsubaih, s.maddock}@dcs.shef.ac.uk

ABSTRACT
Game assets are portable between games. The games themselves are, however, dependent on the game engine they were developed on. Middleware has attempted to address this by, for instance, separating out the AI from the core game engine. Our work takes this further by separating the 'game' from the game engine, and making it portable between game engines. The game elements that we make portable are the game logic, the object model and the game state, which represent the game's brain, and which we collectively refer to as the game factor, or G-factor. We achieve this using an architecture based around a service-oriented approach. We present an overview of this architecture and its use in developing games. The evaluation demonstrates that the architecture does not affect performance unduly, adds little development overhead, is scaleable, and supports modifiability.

Keywords
Game engine, Portability, Game development, Evaluation.

1. INTRODUCTION
The shift in game development from developing games from scratch to using game engines was first introduced by Quake and marked the advent of the game-independent game engine development approach [23]. In this approach, the game engine became "the collection of modules of simulation code that do not directly specify the game's behaviour (game logic) or game's environment (level data)" [26]. The game engine is thus reusable for (or portable to) different game projects. However this shift produces a game that is dependent on the game engine. For example, why can't a player take his favourite game (say Unreal) and play it on Quake engine or Quake game on Unreal engine?

Hardware and software abstractions have facilitated the ability to play a game on different hardware and on different operating systems. These abstractions have also facilitated the ability to use data assets such as 3D models, sound, music, and texture across different game engines. This ability should also be extended to allow for the game itself to be portable. The goal of our work is to make the game engine's brain portable, where the brain holds the game state and the object model and uses the game logic to control the game. We collectively refer to these three things as the G-factor.

We see the portability of the G-factor as the next logical step in the evolution of game development and, following Lewis and Jacobson's terminology [23], we call it the game-engines independent game development approach. A benefit of making the G-factor portable would be to encourage more developers to make use of game engines, since a particular game engine's future capability (or potential discontinuation, as was the fate of Adobe Atmosphere which was used for Adolescent Therapy - Personal Investigator [14]) would not be a worry as a different game engine could easily be substituted. This problem has recently been referred to as "the RenderWare Problem" [11] after the acquisition of RenderWare engine by Electronic Arts (EA) and its removal from the market. We see the issue of rewriting the G-factor from scratch every time we migrate from one engine to another as similar to the undesired practice of developing games from scratch which was deemed unfeasible and resulted in the advent of game engines.

As we noted earlier, portability is an issue that pervades all games with regards to game assets. In addition, however, and related to our work, are the moves towards addressing more aspects of portability. Examples include artificial intelligence (AI) architectures and interfaces [8]. AI architectures use custom made or off-the-shelf components such as AI Middleware (e.g. SOAR [22] or AI.Implant[1]). However, specifying the game using the AI middleware format merely moves the game from one proprietary format (game engines) to another (AI middleware). The work on interfaces aims to facilitate access to game engines. For example, Gamebots [1] and GOLOG Bots [18] are the interfaces that have been used to access Unreal, with, similarly, Quakebot [21] for Quake, FlexBot [20] for Half-Life, and Shadow Door [17] for Neverwinter Nights. These provide interfaces for specific game engines. Other projects are attempting to provide common interfaces to game engines such as the initiative by International Game Developers Association (IGDA) for world interfacing [24] and OASIS [4]. Despite this work, such interfaces may have more success in the serious games community rather than the fast-evolving games industry.

In [9], we described, in detail, how to make the G-factor portable. In this paper, we give an overview of this earlier work, and instead focus more on the evaluation process, addressing issues such as performance, implementation overhead, scalability, and modifiability. We present results of conducting both an unstructured evaluation process and a structured evaluation using ATAM [13], and contrast the two in the subsequent discussion.

The remainder of this paper is structured as follows. Section 2 demonstrates the issues with the typical game development approach through the development of a sample game. This is then contrasted with the development of the same game using our approach, which enables the G-factor to be portable. Section 3 describes the evaluation process and what it revealed about the two development approaches. Finally section 4 presents the conclusions.

2. AN ARCHITECTURE FOR G-FACTOR PORTABILITY
This section contrasts a typical game development approach with the game development approach proposed in our work. Section 2.1 describes what is considered to be a typical development approach through the development of a sample game, and highlights the dependencies associated with this approach. Section 2.2 then proposes an approach to address these dependencies and describes an architecture called game space architecture (GSA) which has been implemented to validate this approach.

[1] http://www.biographictech.com (accessed 5/5/2007).

2.1 A Typical Approach to Game Development

We will use a game we call 'Moody NPCs' to illustrate the typical approach to game development. The game consists of a number of non-player characters (NPCs) that react to a player based on their mood. The player can carry out actions such as greeting or swearing. Each NPC reacts to the action based on his mood which is governed by two variables: cowardness/courage and forgiveness/punishment. The game allows the user to navigate the level and click on an NPC which reveals its current mood and the actions available. The player can adjust the mood variables and try out different actions. The Torque game engine is used to demonstrate how the game is developed.

The typical game development approach can be grouped into four main steps as shown in the typical approach column in Table 1. To create the game level data (step 1), Torque engine provides a level editor called World Editor. The level can also be created using other ways such as: scripting, API, configuration files, etc. The game level data contains the terrain of the environment and the decorative objects (e.g. houses, trees, etc). The level also contains location markers for the game objects (e.g. NPCs and player). Scripting is used to create the other game objects (e.g. Reaction, Action, and Interaction). This approach for creating the game level data is very common amongst game engines - 84% of engines we surveyed provided editors to create the game level [8].

Figure 1 shows the graphical user interface created in step 2. This has mood variables sliders on the top left corner of the screen and an actions controller on the bottom left corner of the screen. The player can use the keyboard to navigate around and the mouse to select an NPC. We used Torque's GUI Editor to set the interface controllers, although it is also possible to use scripting and configuration files.

Step 3 is to create the object model to hold the structure for the game objects. The object model consists of five classes: Player, NPC, Action, Reaction, and Interaction. Torque has a default object model for the player and the AI player. We extended these to add the properties that are specific to the game (i.e. mood variables for an NPC). We created the other classes using a static object model using TorqueScript. The other game object models are created using scripting. Finally, step 4 is to create the game logic which controls how the NPC reacts to the player actions.

Figure 1: The Moody NPCs game.

2.2 GSA's Approach

Figure 2 illustrates the software dependencies problem GSA is aiming to tackle. the example used is the development of 'Gears of War', which is dependent on Unreal Engine 3 and the underlying software [25]. This is similar to the dependency the Moody NPCs game suffers from, and also to the dependencies exhibited by the projects we surveyed in an earlier paper [8].

GSA's objective is to reduce the dependencies by adopting a service-oriented design philosophy, which enables the G-factor to exist independently of the game engine. The service-oriented approach has proved its practicality for achieving different types of portability such as platforms and languages [16]. The novel design approach employed in GSA combines a variant of the model-view-controller (MVC) pattern to separate the G-factor (i.e. model) from the game engine (i.e. view) with on-the-fly scripting to enable communication through an adapter (i.e. controller). The use of a variant of MVC rather than the normal MVC avoids a known liability where the view is tightly coupled to the model [10]. The use of on-the-fly scripting is used to maintain the attractive attributes associated with typical game development where data-driven mechanisms are used to modify the G-factor. Most notably, modifiability is upheld in the typical game development approach using scripting, which our

Table 1: Comparing a typical game development approach to GSA's approach.

Step	Typical Approach	GSA's Approach
1. Create the level data.	• Create the decorative objects in the game engine.	
	• Create the game objects using the world builder or TorqueScript.	• Create the game objects using the world builder in the game engine and give them a unique ID which identifies these objects in the game space as well. Load these objects using TorqueScript. • Create the game objects in the game space with the same unique ID using Jython.
2. Create the GUI.	• Use the game engine interface builder or TorqueScript to create the interface. The behaviour is set as part of the game logic (step 4).	
3. Create the object model.	• Use TorqueScript to extend the objects or create new ones.	• Create the object models for the game objects that require representation in the game engine and the game space. • Create the other game objects models in game space.
4. Create the game logic.	• Use TorqueScript to set the behaviour in the game engine.	• Use Jython or Java to create the logic in the game space.
5. Create the adapter.		• Send the updates from the game engine to the game space. • Create the adapter which translates between the game engine and the game space.

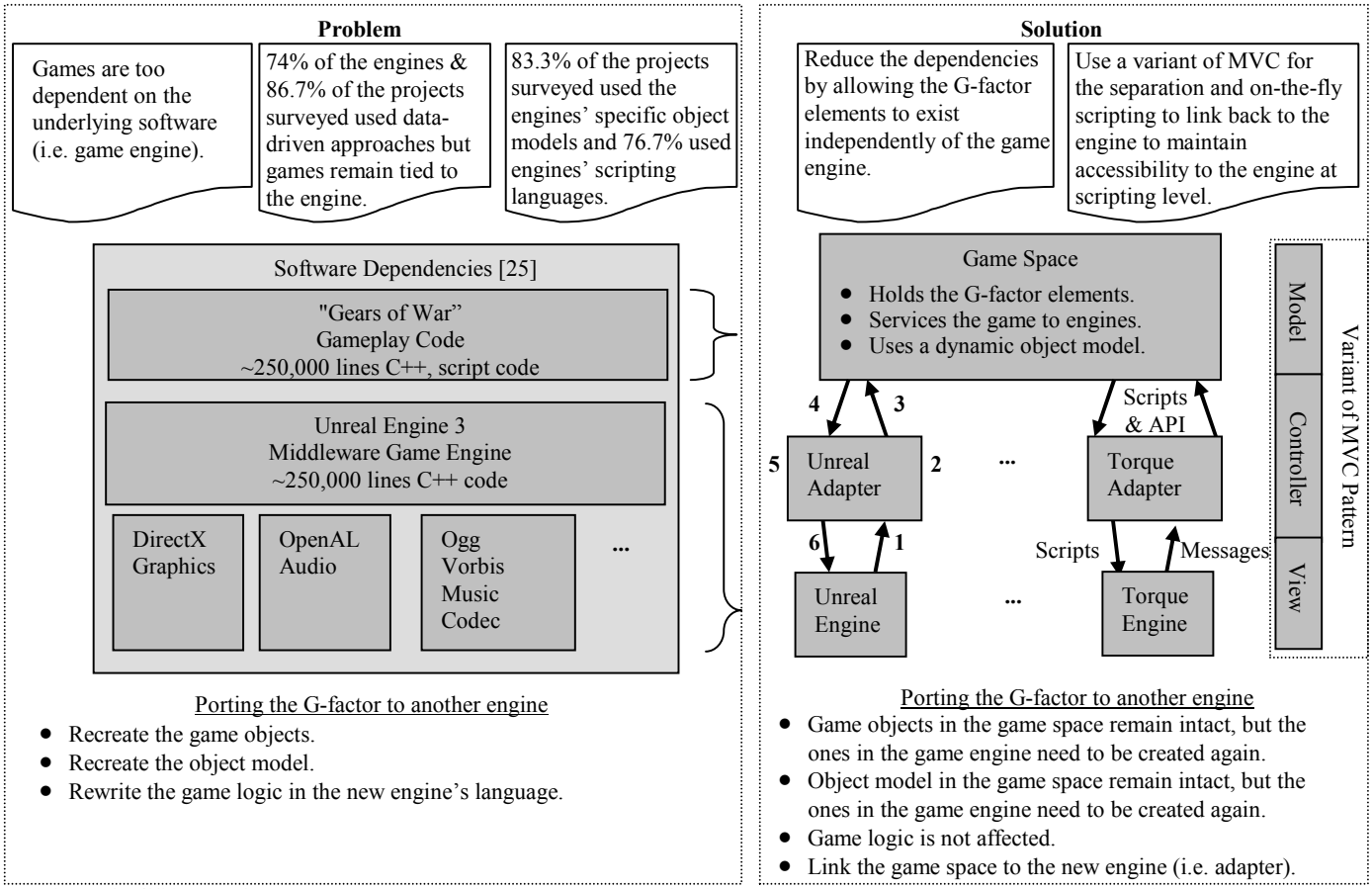

Figure 2: GSA overview. The numbers highlighting the communication between the game space and the game engine are described in Figure 3.

surveys found to be very popular with game engines and projects that use game engines [8]. To maintain this level of modifiability (i.e. scripting level access) to the game engine and the game space, GSA uses on-the-fly scripting to communicate with both via the adapter. For example, a communication may begin with the game engine sending the updates to the adapter (step 1 in the communication protocol shown in Figure 3). The adapter converts them into scripts or direct API calls (step 2) which are then used to update the game space (step 3). When the game space needs to communicate with the game engine, it notifies the adapter of the changes that need to be communicated (step 4). The adapter formats these into the engine's scripting language (step 5) and sends them to the engine to be executed (step 6). The separation and the communication mechanism allow the G-factor to exist independently of the game engine. The effect this has on portability is that when migrating to a new engine the elements in the game space (i.e. the game state, object model, and game logic) can stay intact. Contrasting this to migrating a game developed using the typical game development approach, which often require all three elements to be created again, shows the extent of the effort saved.

As was shown in Table 1, the first difference between our approach and the typical game development approach is the creation of the game objects, which is split over the game engine and the game space due to the two types of game objects. The first type are the game objects that have to have representations inside the game engine to provide visual representations, such as the Player and the NPCs needed for the Moody NPCs game. These require real-time processing in the game engine and it is impractical to communicate every frame from the game space to the game engine. Therefore these objects have to be created in the game engine as well as the game space and only updates are communicated. The second type of game objects are the ones that do not have representations inside the game engine, such as the Action, Interaction, and Reaction objects. These objects can be created in the game space only. The object model creation is similarly split over the game engine and the game space. The second difference is creating the game logic in the game space

1. Updates are received from the game engine.
2. The adapter uses the scripts mapping table to convert the message to a Jython script.
3. Game state is updated.
4. When a modification is done in the game state the adapter is notified.
5. If the object is of class interest then the adapter converts it to a game engine script.
6. Script is sent to the game engine.

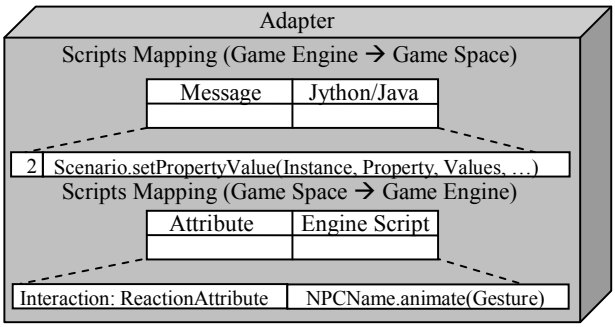

Figure 3: Communication between the game engine and the game space.

rather than the game engine. The third difference is creating the adapter which handles the communication between the game space and the game engine.

3. EVALUATION AND DISCUSSION

A software architecture can be evaluated using structured or unstructured methods. An unstructured evaluation, which is a common way to evaluate a software architecture [3], consists of randomly throwing challenges at the architecture and hoping that either the architecture can address them, or that they will reveal its limitations. In structured evaluation, methods such as ATAM [13], SAAM [19], ARID [12], and ABAS, PASA and CBAM [2] are used to probe the architecture with the aim of exercising the whole architecture. We used ATAM in our structured evaluation, a method that is not limited to a particular stage of the development cycle, and which involves stakeholders (i.e. user, maintainer, developer, manager, tester, architect, security expert, etc.) in specifying the architecture attributes to address. In the following paragraphs, we will summarise the findings of detailed structured [6] and unstructured [5] evaluations carried out in our earlier research papers. We will focus on four attributes: portability, performance, modifiability and scalability. Following this, we contrast the structured and unstructured approaches to evaluation.

3.1 Portability

The unstructured evaluation found that GSA managed to address the portability challenge by servicing the same G-factor to two different engines [4]: a bespoke engine developed on top of DirectX 9.0 and the Torque game engine (see Figure 4). This was done without modifying the G-factor and was constrained to modifying the adapter. Similarly, the structured evaluation found GSA supports portability. It found that the separation using the MVC pattern allows for better portability since it allows for multiple views (i.e. game engines) for the same model (i.e. G-factor). In addition, the structured evaluation found that portability could be undermined if the game engine does not fully expose the required functionality through scripting since the adapter relies on scripting for communicating back to the game engine (see Figure 2).

3.2 Performance

The aim here was to find the average reduction in frames-per-second (fps) due to the use of GSA. To get a performance indicator a player was simulated to be running continuously around a path for 30 minutes (see Figure 5). Using this simulation, two performance tests were run to contrast the overheads of a game developed with the typical development approach to one developed using GSA. The performance overheads measured were: fps, CPU, memory, and network (for the test using the game space). The average reduction in fps was 11.69% when following the GSA approach. This average fps reduction is relatively large for a small game and more tests need to be performed to get a better indication of how this reduction will scale with the game size. However, when comparing this finding to the findings from the scalability challenge (described later) we find that GSA does not affect performance unduly. The structured evaluation revealed two issues. It found that the data integrity across the different game states (i.e. game engine and game space) was at risk. This is due to the delays that might occur because of the separation as a result of the use of the MVC pattern which add an overhead for exchanging information. Initial tests revealed no problems, but further tests are required before this can be established with certainty. In addition, there is a danger if the message load increases that the game space becomes the bottleneck in the architecture.

3.3 Modifiability

Here, the success of GSA was judged by the ability to create different G-factors on the same architecture using a different object model and game logic. The fact that different G-factors (Figure 1, Figure 4, Figure 5, and Figure 6) can be developed using GSA showed its modifiability. In addition, a structured evaluation process measured the modifiability across the different parts of GSA by examining how each architectural decision affects modifiability and how it trades against the other quality attributes (e.g. portability and performance) [6]. The evaluation revealed that if a single unique identifier cannot be set for game objects on the game space and game engine then GSA becomes very sensitive to any modification as it has to be added manually in the adapter. Furthermore, using on-the-fly scripting allows for better modifiability but runs slower than pre-compiled code. Modifiability is also enhanced by the use of a variant of the MVC pattern that reduces the dependencies between the model and the view.

3.4 Scalability

The aim was to identify how much overhead is added as the game size grows. This was examined by developing a serious game for traffic accident investigators [7] (see Figure 6). The adapter's implementation overhead for each challenge is presented in Table 2. Using the implementation overheads of the adapters compared to their game logic sizes in each of the test games developed, we can forecast that for small game size the overhead is large, but that it stabilises at around 6% for code of size between 100,000 and 500,000 lines[2]. The scalability challenge also showed that performance overhead was not noticeable when judging its success in training [7] for which smooth play is crucial to avoid frustrating the users. The structured evaluation found that using a dynamic object model allows for better game model scalability but it makes the architecture very sensitive to change as the change propagates to the game logic and to the adapter.

Figure 4: (left) smart terrain running on bespoke engine; (right) the same G-factor running on Torque [5].

Figure 5: First-person shooter game [5].

Figure 6: A serious game for traffic accident investigators [7].

[2] http://support.microsoft.com/kb/828236 (accessed 24/8/2007).

Table 2: The implementation overhead for the adapter.

Challenge	Logic size (lines of code)	Adapter size (lines of code)
Portability	60	346 (bespoke) 354 (Torque)
Modifiability	100	350
Performance	70	300
Scalability	6214	1100

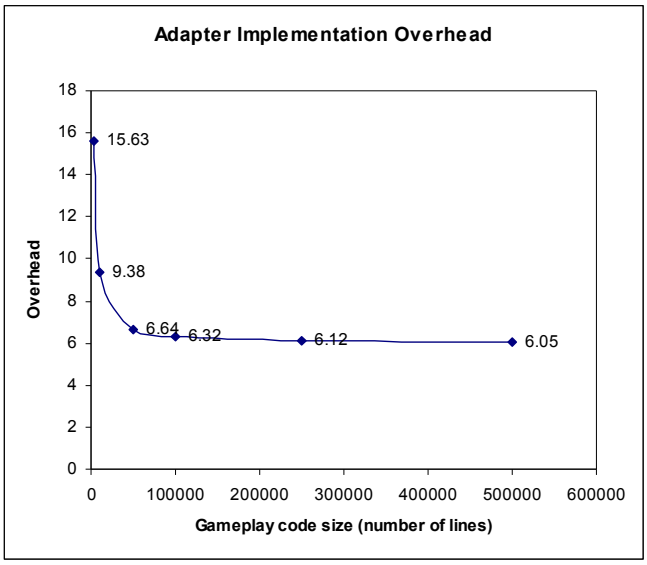

Figure 7: The adapter code forecast compared to the game logic code size.

3.5 Structured vs. Unstructured Evaluation

The unstructured evaluation revealed how well the architecture can cope with the challenges. However, there was no easy way to establish the correlation between the challenge and what architectural decisions had supported or undermined it. Furthermore the unsystematic way of generating scenarios (i.e. challenges) meant that some time was unnecessarily spent in implementing different tests when one could have served all the challenges (e.g. the implementation of the serious game (see Figure 6) used in the scalability challenge could have been used to test all of the challenges). This could be attributed to the incomplete overall evaluation picture due to the lack of systematic guidance. Although, there is no guarantee that a structured evaluation would not produce redundant probing since, just like the unstructured evaluation, it is also scenario-based. However, the chances are reduced due to the fact that the generation of scenarios is guided by using a utility tree[3] in which all the scenarios are identified. This serves two purposes. The first purpose is that once all the scenarios are present the experimentation can begin by choosing a test where preferably all these scenarios can be addressed. The second purpose is that it describes the decisions that are going to be analyzed by the scenario which means any repetitive probing can be identified.

The problem with scenario-based evaluation which both unstructured and structured evaluations use is that the evaluation is only as good as the scenarios generated, which in turn depends on the stakeholders in the evaluation team. Although there are measures put in place to ensure the selection includes all the important personnel (i.e. architects and domain experts), the fundamental problem still persists.

Contrasting ATAM's output to the unstructured evaluation results, which quite often answer the challenge with yes or no, or with some metrics such as network load or fps, highlights the strengths of ATAM. ATAM classifies the decisions according to how they affect the architecture (i.e. support or undermine it). We found the ATAM process helpful in understanding our architecture better. Of further benefit is that it should also act as a guide when there is a need to modify or evolve GSA. This guidance is based on the fact that it reveals the strengths and weaknesses of the architectural decisions. In future, we recommend using ATAM alongside the development cycle. This is where ATAM is designed to be most effective by revealing issues at different stages of the development cycle when they are cheaper to address. Had we started with ATAM we believe it would have saved us time and effort by avoiding the creation of a number of redundant challenges.

4. CONCLUSIONS

We have presented an architecture for making 'games' (i.e. G-factors) portable between game engines. The changes required to the typical game development approach have been demonstrated through the development of a sample game called Moody NPCs. In addition, the work has presented the findings from two types of evaluation. The findings have revealed that GSA is capable of making the G-factor portable, but GSA adds performance and implementation overheads. Despite these overheads, GSA has been shown to scale to real world applications [7]. Modifiability has been found to be sensitive in cases where a unique identifier cannot be set for game objects.

Whilst the unstructured evaluation managed to reveal issues with the architecture, the mechanism of throwing random challenges resulted in redundant challenges and failed to articulate which architectural decisions undermined or supported GSA. Using ATAM guided the evaluation better. Employed earlier, it could have helped to avoid the redundancy in the unstructured evaluation. Also, it was capable of revealing how the architectural decisions interact in order to support the required attributes. Although the portability presented in this work has only been shown across two engines, the approach followed to achieve that is consistent in the way the two engines were linked via the adapter and therefore there is no reason why it cannot be followed to link other engines.

With gameplay predicted to be the distinguishing factor between future games [15] and combined with the increased number of commercial licensees of game engines and the interest engines are receiving from outside the games industry (e.g. the serious games community), this will increase the need for portable games for two reasons. The first reason is because developers can keep the visual aspects of their game up to date with the latest game engine. The second reason is the security from having to face 'the RenderWare Problem' [11]. However, the incentive for game engine developers is less clear.

5. REFERENCES

[1] Adobbati, R., Marshall, A.N., Scholer, A., Tejada, S., Kaminka, G., Schaffer, S., and Sollitto, C. (2001). Gamebots: a 3D virtual world test-bed for multi-agent research. In *Proceedings of 2nd International Workshop on Infrastructure, MAS and MAS Scalability*.

[3] The utility tree elicits the quality attributes down to the scenario level to provide a mechanism for translating architectural requirements into concrete practical scenarios.

[2] Bahsoon, R., and Emmerich, W. (2003). Evaluating software architectures: development, stability and evolution. In *Proceedings of ACS/IEEE International Conference on Computer Systems and Applications*.

[3] Bahsoon, R., and Emmerich, W. (2006). Architectural Stability and Middleware: An Architecture Centric Evolution Perspective. In *Proceedings of the ECOOP 2006 workshop on Architecture-Centric Evolution*.

[4] Berndt, C., Watson, I., and Guesgen, H. (2005). OASIS: an open AI standard interface specification to support reasoning, representation and learning in computer games. In *Proceedings of Proceedings of Workshop for International Joint Conference on Artificial Intelligence (IJCAI-05), Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 19-24.

[5] BinSubaih, A., Maddock, S., and Romano, D. (2005). Game logic portability. In *Proceedings of ACM SIGCHI International Conference on Advances in Computer Entertainment Technology ACE 2005*, *Computer Games Technology session*, pages 458-461.

[6] BinSubaih A., and Maddock S. (2006). Using ATAM to evaluate a game-based architecture. *Workshop on Architecture-Centric Evolution (ACE 2006)*, *hosted at the 20th European Conference on Object-Oriented Programming ECOOP 2006*.

[7] BinSubaih A., Maddock S., and Romano D. (2006). A serious game for traffic accident investigators. Special Issue of International Journal of Interactive Technology and Smart Education on "Computer Game-based Learning.", 3(4):329-346.

[8] BinSubaih, A., Maddock, S., and Romano, D. (2007). A survey of 'game' portability. Department of Computer Science Technical Report CS-07-05, 2007, University of Sheffield.

[9] BinSubaih, A., and Maddock, S. (2007). G-factor portability in game development using game engines. In *Proceedings of The Third International Conference on Games Research and Development 2007*.

[10] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). Pattern-oriented software architecture: a system of patterns. Volume 1, John Wiley and Sons.

[11] Carless, S. (2007). Rise of the game engine. *Game Developer*, April, 2007, pages 2-2.

[12] Clements, P. (2000). Active reviews for intermediate designs. Technical Report CMU/SEI-2000-TN-009, Software Engineering Institute.

[13] Clements, P., Kazman, R., and Klein, M.(2001). *Evaluating software architectures: methods and case studies*, Addison Wesley.

[14] Coyle, D. and Matthews, M. Personal Investigator: a Therapeutic 3D Game for Teenagers. *CHI2004* Vienna 25-29 April 2004. Presented at the Social Learning Through Gaming Workshop.

[15] Dounis, E. (2006). The great debate: gameplay vs. graphics, GamersMark.com,7/9/2006,http://www.gamersmark.com/articles/205/ (accessed 31/1/2007).

[16] Erl, T. (2005). *Service-oriented architecture: concepts, technology, and design*. Prentice Hall.

[17] Hussain, T.S., and Vidaver, G. (2006). Flexible and purposeful NPC behaviors using real-time genetic control. In *Proceedings of the 2006 World Congress on Computational Intelligence*, pages 785-792.

[18] Jacobs, S., Ferrein, A., and Lakemeyer, G. (2005). Unreal Golog Bots. In *Proceedings of Workshop for International Joint Conference on Artificial Intelligence (IJCAI-05), Workshop on Reasoning, Representation, and Learning in Computer Games*.

[19] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, J. (1998). The architecture tradeoff analysis method. In *Proceedings of Fourth IEEE International Conference on Engineering of Complex Computer Systems*, pages 68-78.

[20] Khoo, A., Dunham, G., Trienens, N., and Sood, S. (2002). Efficient, realistic NPC control systems using behavior-based techniques. In *Proceedings of the AAAI 2002 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment*.

[21] Laird, J. (2001). It knows what you're going to do: adding anticipation to a Quakebot. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 385-392.

[22] Laird, J.E., Assanie, M., Bachelor, B., Benninghoff, N., Enam, S., Jones, B., Kerfoot, A., Lauver, C., Magerko, B., Sheiman, J., Stokes, D., and Wallace, S. (2002). A testbed for developing intelligent synthetic characters. In *Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Spring Symposium,* pages 52-56.

[23] Lewis, M., and Jacobson, J. (2002) Game engines in scientific research. *Communications of the Association for Computing Machinery (CACM)*, 45(1):27-31.

[24] Nareyek, A., Combs, N., Karlsson, B., Mesdaghi, S., and Wilson, I. (2005). The 2005 report of the IGDA's artificial intelligence interface standards committee. International Game Developers Association, http://www.igda.org/ai/report-2005/report-2005.html (accessed 14/8/2007).

[25] Sweeney T. (2006). The Next Mainstream Programming Language: A Game Developer's Perspective. The 33rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages Charleston, South Carolina. www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt (accessed 5/5/2007).

[26] Wang, J., Lewis, M., and Gennari J. (2003). Emerging areas: urban operations and UCAVs: a game engine based simulation of the NIST urban search and rescue arenas. In *Proceedings of the 35th Winter Simulation Conference,* pages 1039-1045.